Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 9: this Friday @ 11:59 pm
  - Lab 8: Tuesday 3/28 @ 11:59 pm
  - Assignment 3: Friday 3/31 @ 11:59 pm
    - Support video and slides on Canvas

Previous lecture

- ADT Graph
  - definitions
  - representations
  - traversals
    - BFS

This Lecture

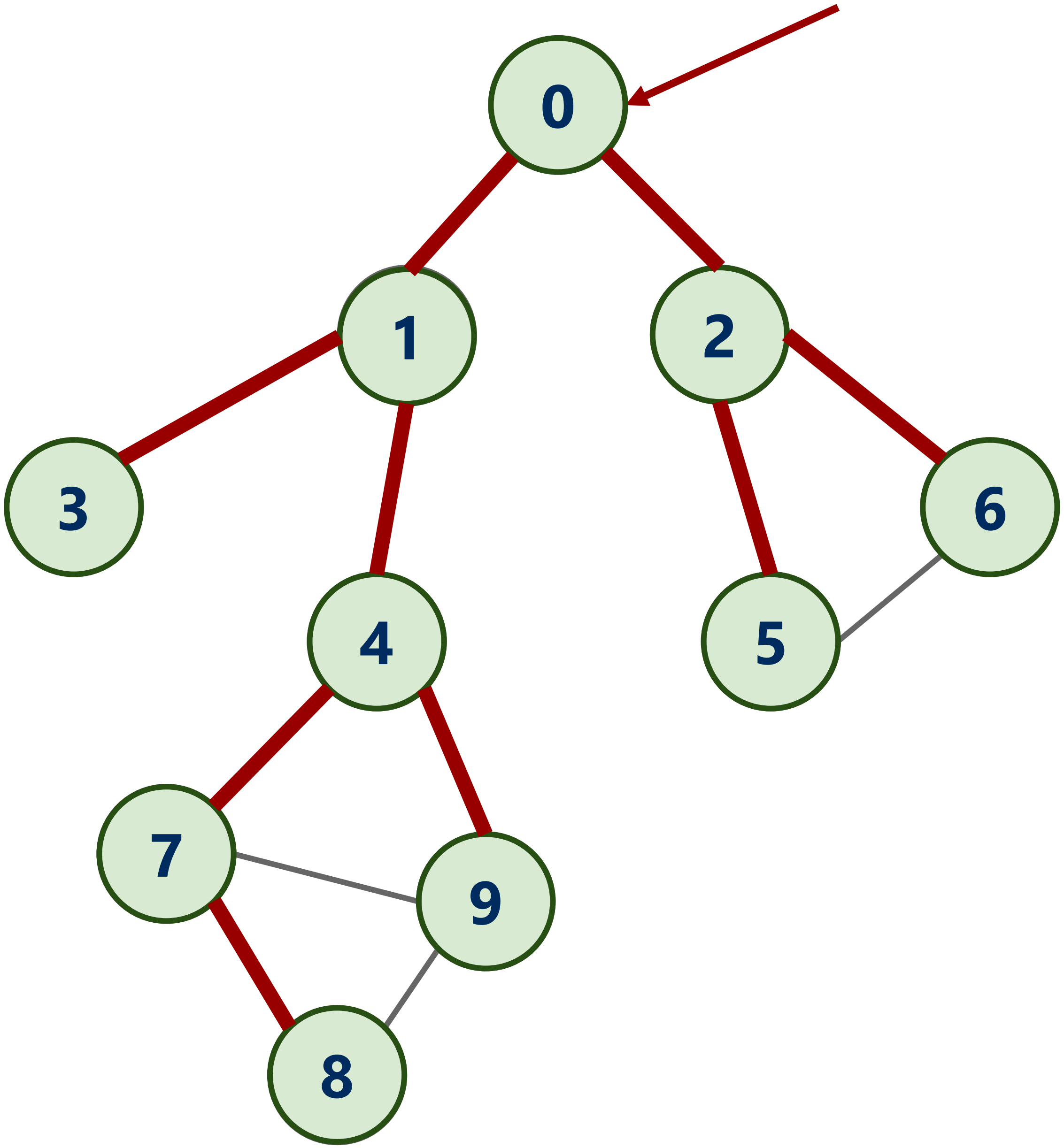
- ADT Graph
  - traversals
    - BFS
      - shortest paths based on number of edges
      - connected components
    - DFS
      - finding articulation points of a graph
  - Minimum Spanning Tree (MST) problem
    - Prim’s MST algorithm

BFS Pseudo-code

```
Q = new Queue

BFS(vertex v){
    add v to Q
    while(Q is not empty){
        w = remove head of Q
        visited[w] = true //mark w as visited
        for each unseen neighbor x
            seen[x] = true //mark x as seen
            parent[x] = w
            add x to Q
    }
}
```

BFS example



Shortest paths

- BFS traversals can further be used to determine the *shortest path* between two vertices

BFS Pseudo-code to compute shortest paths

```
Q = new Queue

BFS(vertex v){
    add v to Q
    while(Q is not empty){
        w = remove head of Q
        visited[w] = true //mark w as visited
        for each unseen neighbor x
            seen[x] = true //mark x as seen
            parent[x] = w
            distance[x] = distance[w] + 1
            add x to Q
    }
}
```

Problem of the Day

- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …

- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1st connection, 2nd connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

Problem of the Day

- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …

- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1st connection, 2nd connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

Finding connected components

- A connected component is a connected subgraph G′
  - (V′, E′)
    - V′ ⊆ V
    - E′ = {(u, v) ∈ E and both u and v ∈ V′}
- To find all connected components:
  - wrapper function around BFS
  - A loop in the wrapper function will have to continually call bfs() while **there are still unseen vertices**
  - Each call will yield a spanning tree for a **connected component** of the graph

BFS Pseudo-code to compute connected components

```
int components = 0
for each vertex v in V
        if visited[v] = false
                components++
                Q = new Queue
                BFS(v)
```

```
BFS(vertex v){
        add v to Q
        component
        while(Q is not empty){
                w = remove head of Q
                visited[w] = true
                component[w] = components
                for each unseen neighbor x
                        seen[x] = true
                        add x to Q
        }
}
```

Problem of the Day

- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …

- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1st connection, 2nd connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

Runtime Analysis of BFS

- Total time: **vertex processing time + edge processing tim**e
- Each vertex is added to the queue exactly once and removed exactly once
  - *v* add/remove operations
    - *O(v)* time for vertex processing
- Edges are processed when adding the list of neighbors to the queue

Runtime Analysis of BFS: Adjacency Lists

- Each edge is processed at most twice, one per edge endpoint
  - *O(e)* time for edge processing
- Total time: **vertex processing time + edge processing tim**e
  - *O(v + e)*

Runtime Analysis for BFS: Adjacency Matrix

- With Adjacency Matrix, BFS checks each *possible* edge!
    - $O(v^2)$ time for edge processing
- Total time: **vertex processing time + edge processing time**
    - $O(v^2 + v) = O(v^2)$
- ***Running time depends on data structure selection!***

Problem of the Day

- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …

- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1st connection, 2nd connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

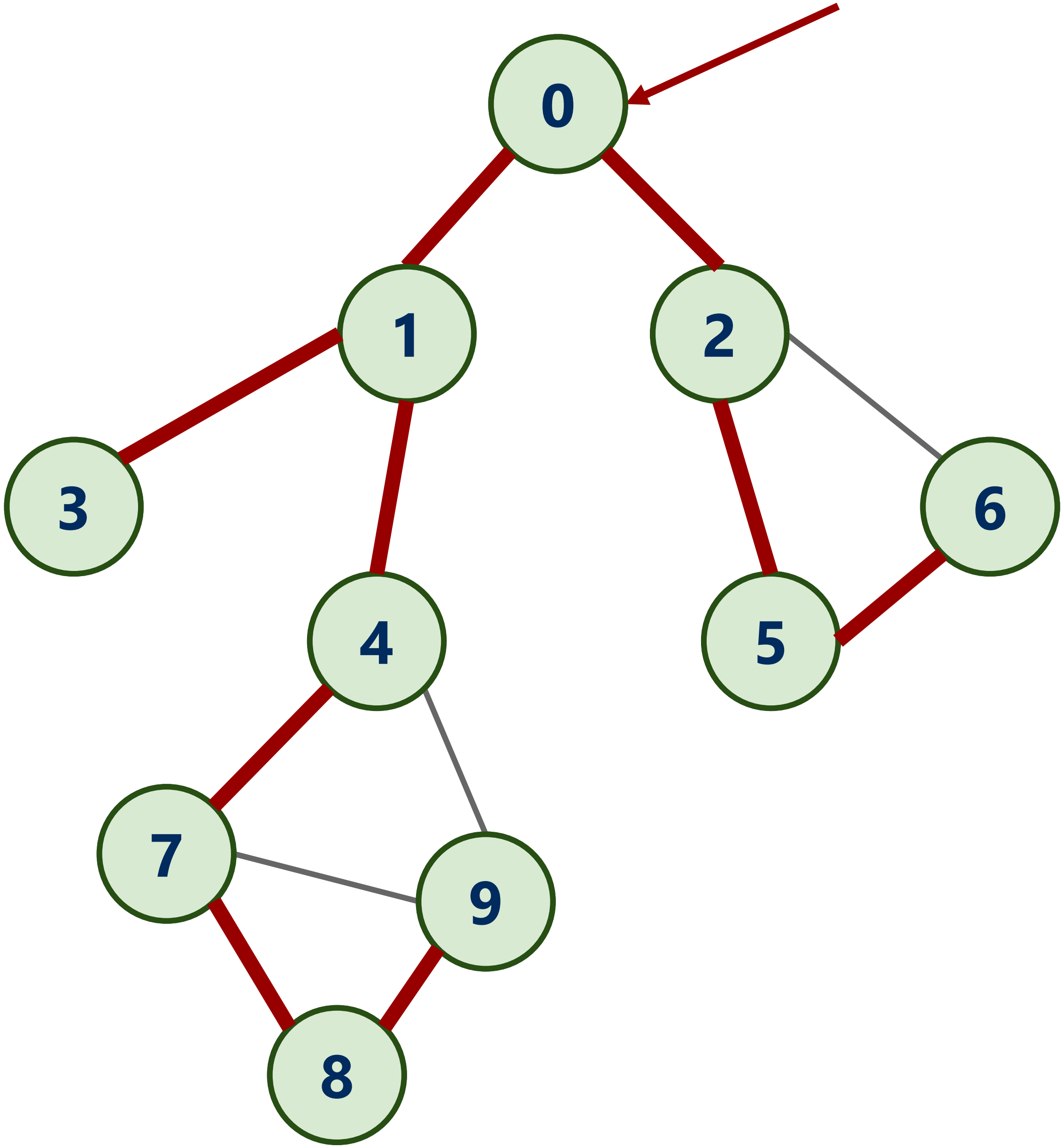
DFS – Depth First Search

- Already seen and used this throughout the term
  - For Huffman encoding…
    - as we build the codebook from the Huffman Trie
- Can be easily implemented recursively
  - For each vertex, visit *first* unseen neighbor
  - Backtrack at deadends (i.e., vertices with no unseen neighbors)
    - Try *next* unseen neighbor after backtracking
  - An arbitrary order of neighbors is assumed

DFS Pseudo-code

```
DFS(vertex v) {
  seen[v] = true //mark v as seen
  for each unseen neighbor w
    parent[w] = v
    DFS(w)
}
```

DFS example



9
8
6
5
2
0

Runtime Stack

When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
   visit v //pre-order DFS
   for each unseen neighbor w
      parent[w] = v
      DFS(w)
}
```

When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
for each unseen neighbor w
      parent[w] = v
      DFS(w)
 visit v //post-order DFS
}
```

When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
for each unseen neighbor w
      parent[w] = v
      DFS(w)
      (re)visit v //in-order DFS
}
```

Runtime Analysis of DFS: Adjacency Lists

- Total time: **vertex processing time + edge processing time**
- Each vertex is seen then visited exactly once
  - *O(v)* time for vertex processing
  - except for in-order DFS
    - vertex processing is included in edge processing in that case
- Edges are processed when finding the list of neighbors
- Each edge is checked at most twice, one per edge endpoint
  - *O(e)* time for edge processing
- Total time: *O(v + e)*

Runtime Analysis of BFS and DFS

- At a high level, DFS and BFS have the same runtime
  - Each vertex must be seen and then visited, but the order will differ between these two approaches
- The representation of the graph affect the runtimes of of these traversal algorithms?
  - $O(v + e)$ with Adjacency Lists
  - $O(v^2)$ with Adjacency Matrix
  - Note that for a dense graph, $v + e = O(v^2)$

Problem of the Day

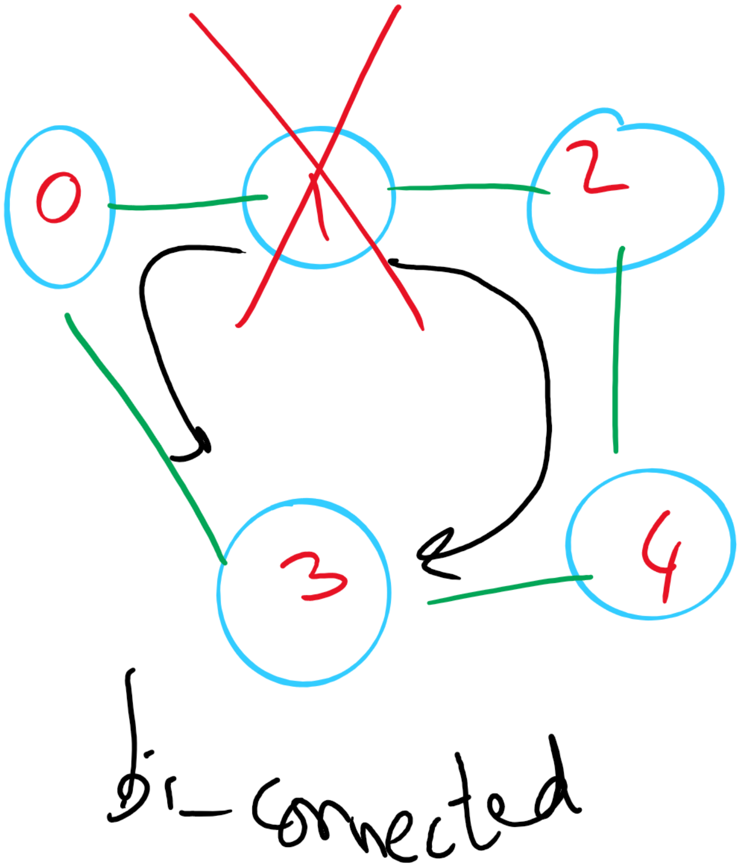
- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …

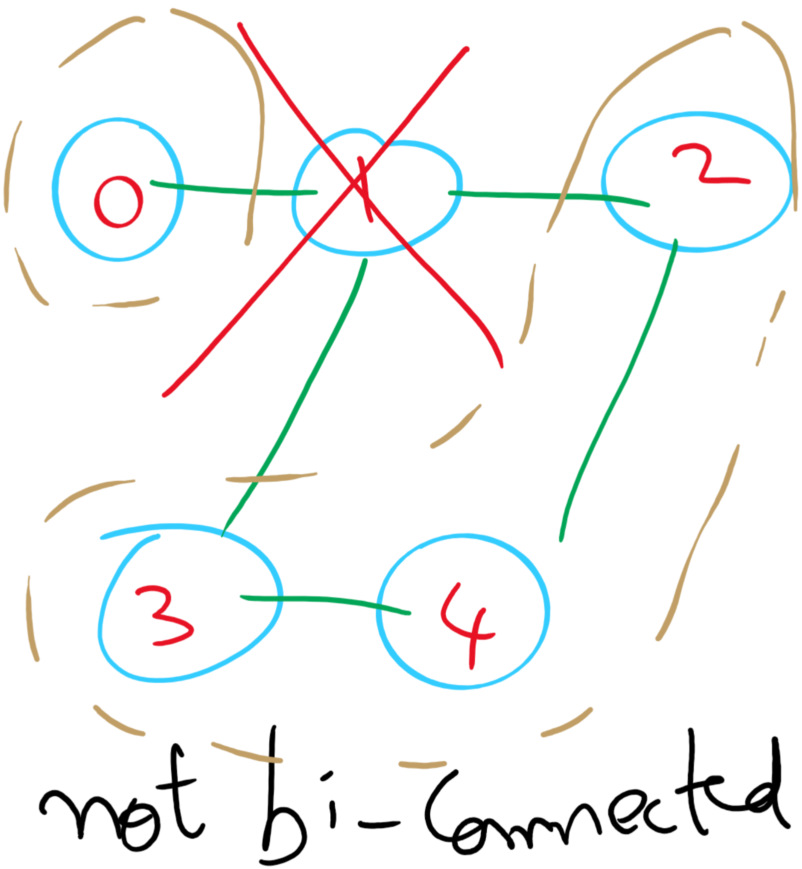
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1st connection, 2nd connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

Biconnected graphs

- A *biconnected graph* has at least 2 distinct paths between all vertex pairs
  - a distinct path shares no common edges or vertices with another path except for the start and end vertices
- A graph is biconnected graph iff it has zero *articulation points*
  - Vertices, that, if removed, will separate the graph

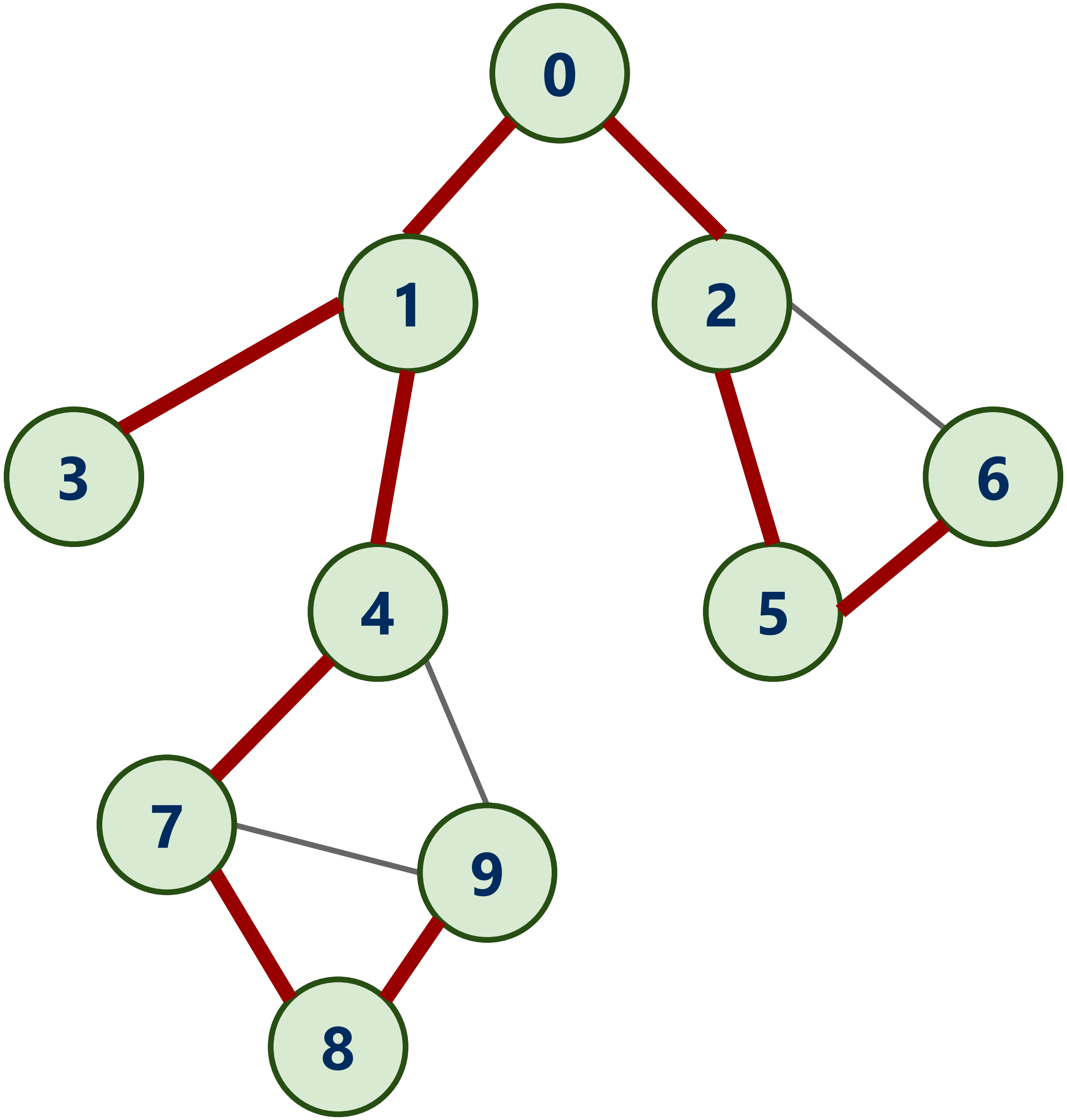
Biconnected Graph





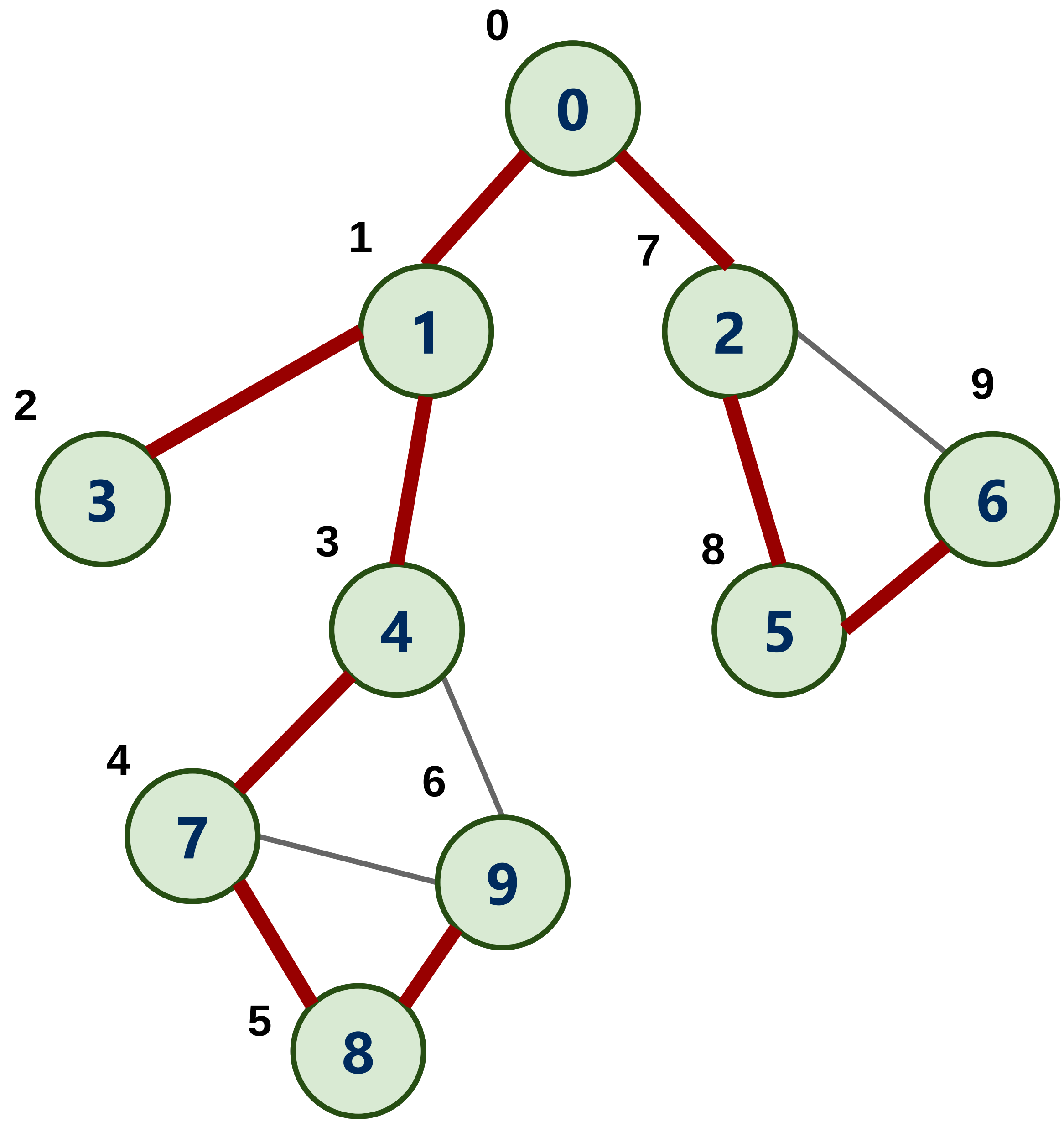
Finding articulation points of a graph

- A DFS traversal builds a spanning tree
  - red edges in the picture
- Edges not included in the spanning tree are called **back edges**
  - e.g., (4, 9) and (2, 6)



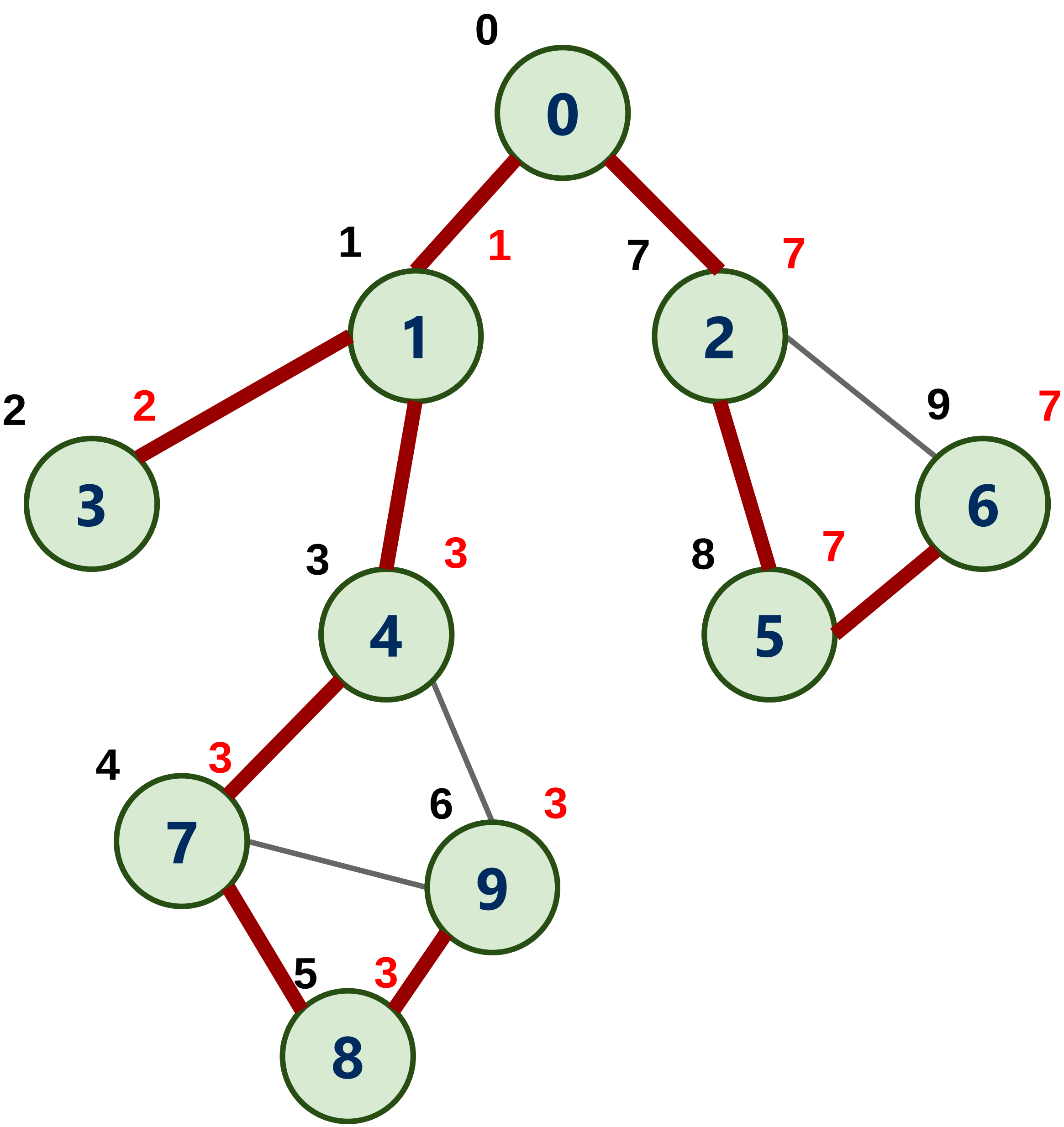
num(v)

- A pre-order DFS traversal visits the vertices in some order
  - let's number the vertices with their traversal order
  - num(v)



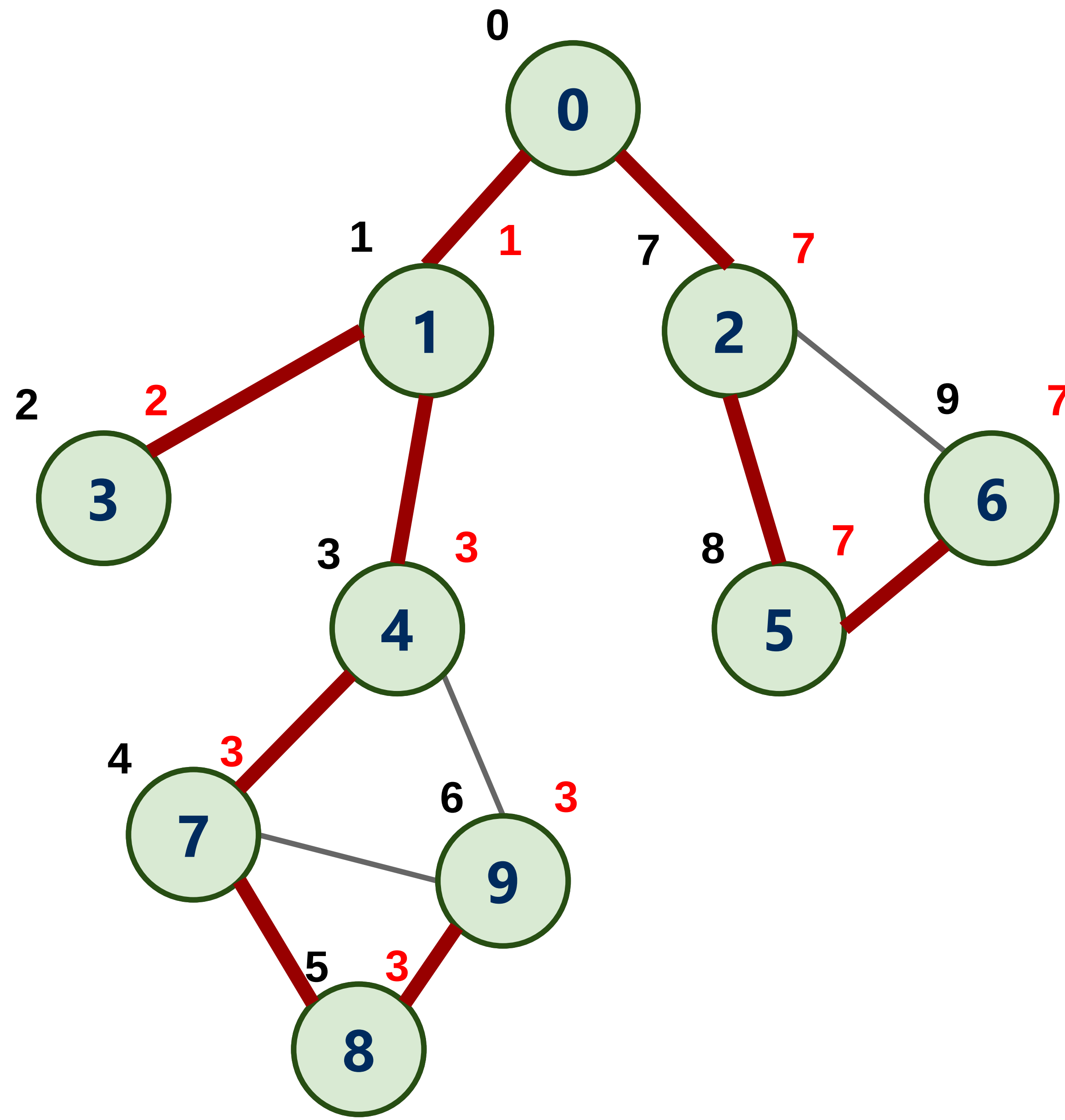
Finding articulation points of a graph

- For each non-root vertex v, find the lowest numbered vertex reachable from v
  - **not through v's parent**
  - **using 0 or more tree edges then at most one back edge**
- move down the tree looking for a back edge that goes backwards the furtheset



low(v)

- How do we find low(v)?
- low(v) = Min of:
  - num(v)
  - num(w) for all back edges (v, w)
  - low(w) of all children of *v*

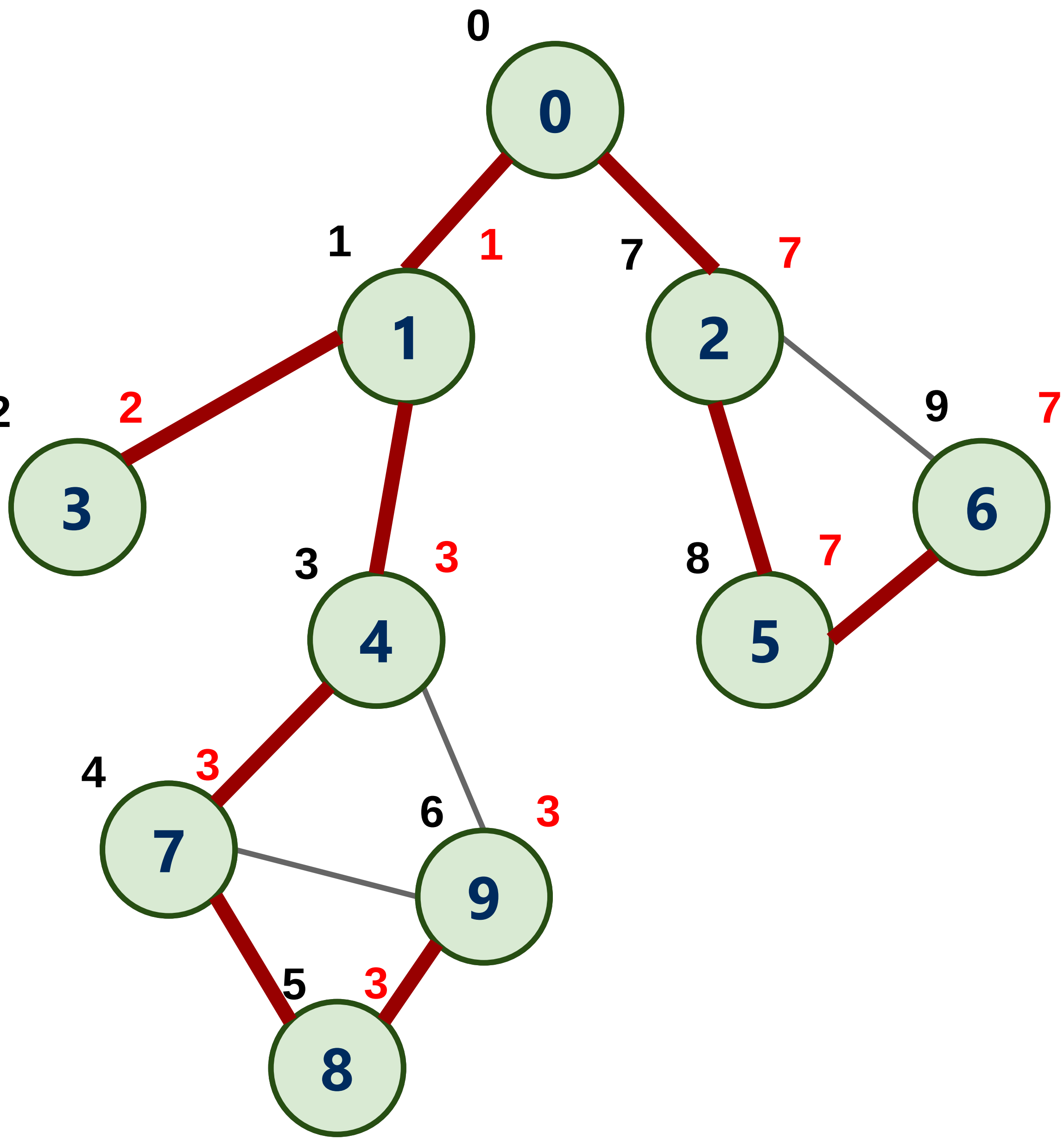


low(v)

- low(v) = lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then **at most one** back edge
  - Min of:
    - num(v) (the vertex is reachable from itself)
    - Lowest num(w) of all back edges (v, w)
    - Lowest low(w) of all children of *v* (the lowest-numbered vertex reachable through a child)

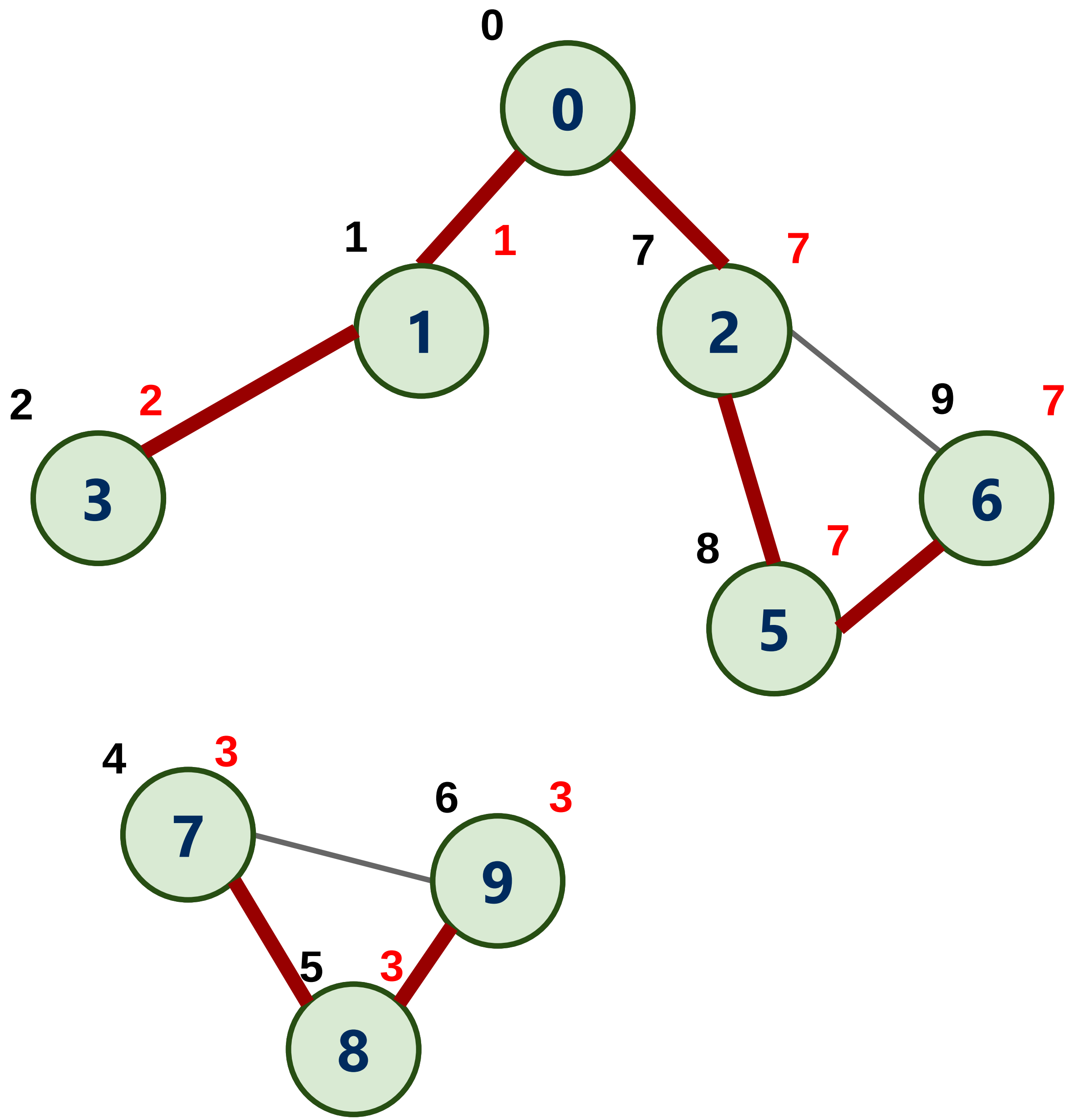
Why are we computing low(v)?

- What does it mean if a vertex has a child such that
  - **low(child) >= num(parent)**?
- e.g., 4 and 7
- child has **no other way** except through parent to reach vertices with lower num values than parent
- e.g., 7 cannot reach 0, 1, and 3 except through 4
- So, the **parent is an articulation point**!
  - e.g., if 4 is removed, the graph becomes disconnected



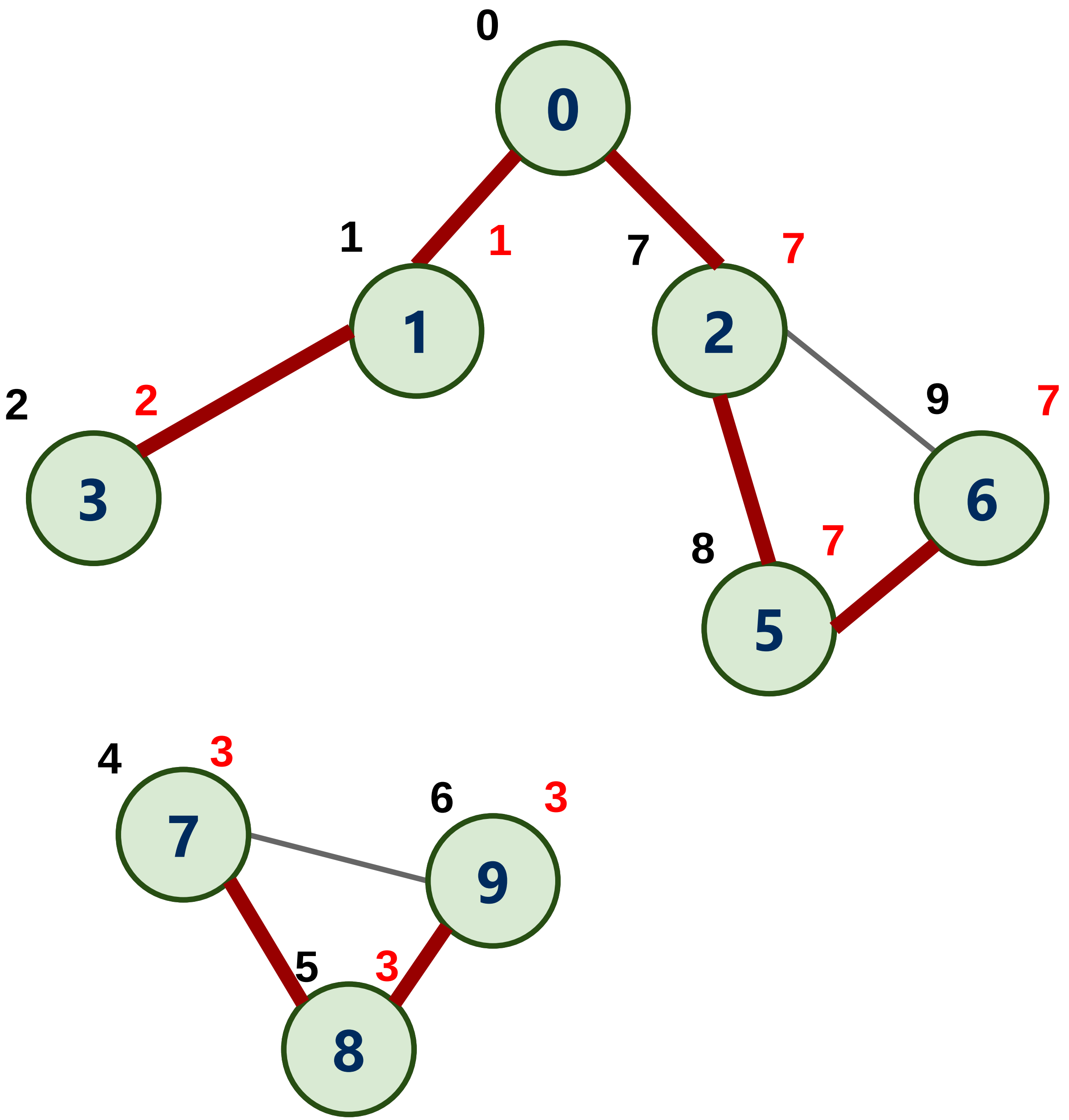
Why are we computing low(v)?

- if 4 is removed, the graph becomes disconnected
- Each **non-root vertex v** that has a child w such that **low(w) >= num(v) is an articulation point**



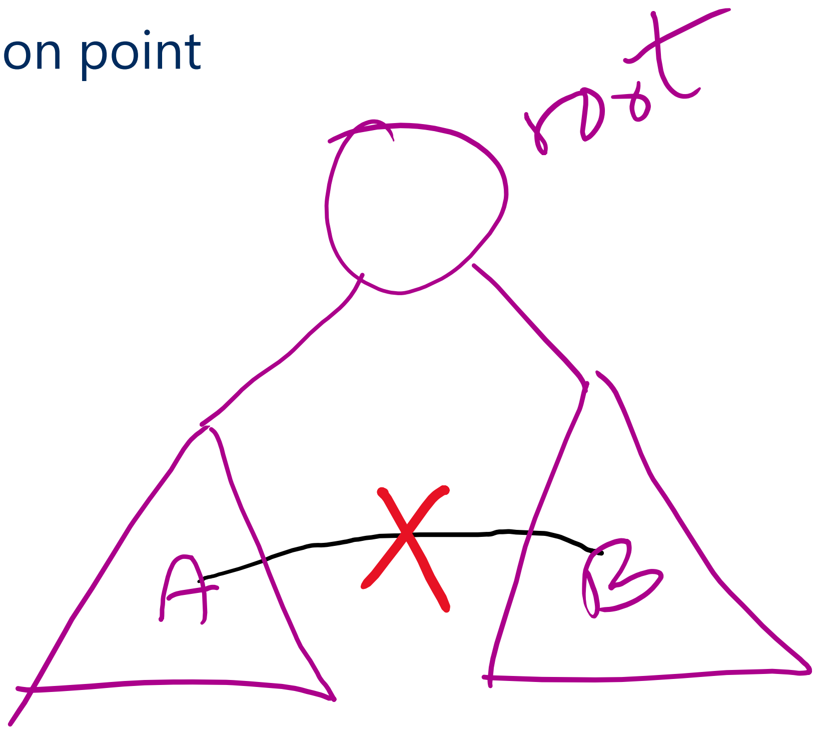
What about the root vertex?

- The root has the smallest num value
  - root's children can't go "further" than root
- Possible that low(child) == num(root) but root is not an articulation point
- need a different condition for root



What about the root of the spanning tree?

- What if we start DFS at an articulation point?
  - The starting vertex becomes the root of the spanning tree
  - If the root of the spanning tree has more than one child, the root is an articulation point



Finding articulation points of a graph: The Algorithm

- As DFS visits each vertex v
  - Label v with with the two numbers:
    - num(v)
    - low(v): initial value is num(v)
  - For each neighbor w
    - if already seen → we have a back edge
      - update low(v) to num(w) if num(w) is less
    - if not seen → we have a child
      - call DFS on the child
      - **after the call returns,**
        - update low(v) to low(w) if low(w) is less

when to compute num(v) and low(v)

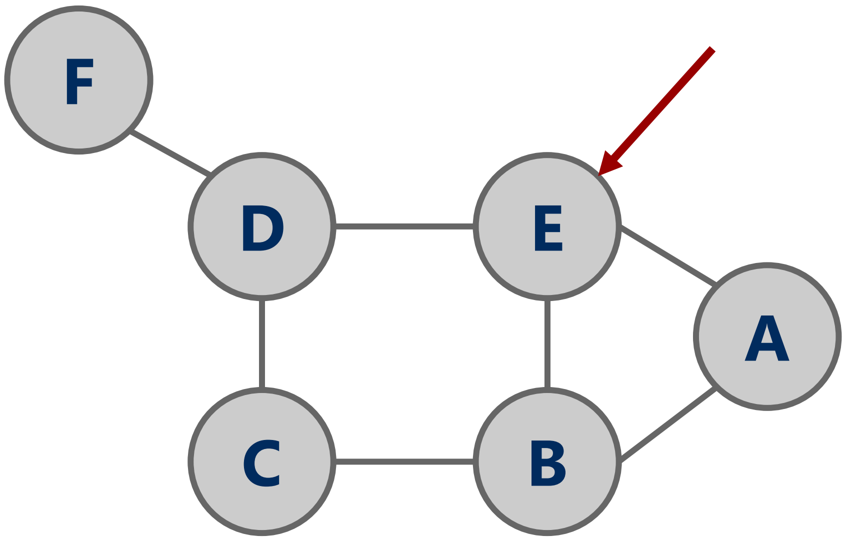
- num(v) is computed as we move down the tree
  - pre-order DFS
- low(v) is updated as we move down and up the tree
- Recursive DFS is convenient to compute both
  - why?

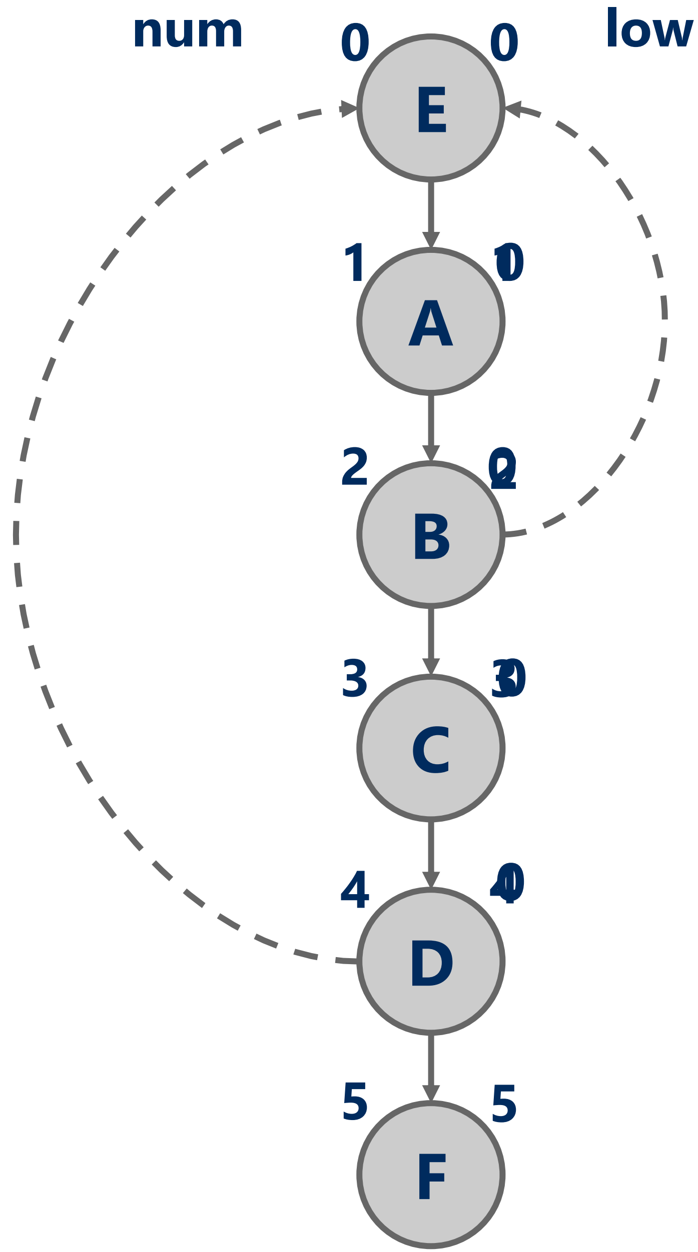
Using DFS to find the articulation points of a connected undirected graph

```
int num = 0

DFS(vertex v) {
    num[v] = num++
    low[v] = num[v] //initially
    seen[v] = true //mark v as seen
    for each neighbor w
       if(w unseen){
          parent[w] = v
          DFS(w) //after the call returns low[w] is computed, why?
          low[v] = min(low[v], low[w])
          if(low[w] >= num[v]) v is an articulation point
       } else { //seen neighbor
          if(w != parent[v]) //and not the parent, so back edge
             low[v] = min(low[v], num[w])
       }
```

Finding articulation points example





Neighborhood connectivity Problem

- We want to keep a set of neighborhoods connected with the minimum cost possible
- **Input:** A set of neighborhoods and a file with the following format:
    - neighborhood i, neighborhood j, cost of connecting the two neighborhoods
    - …
- **Output:** A set of neighborhood pairs to be connected and a total cost such that
    - We can go from any neighborhood to any other **(connected)**
    - The total cost should be minimum (i.e., as small as it can be) **(minimal cost)**

Think Data Structures First!

- How can we structure the input in computer memory?
- Can we use Graphs?
- What about the costs? How can we model that?

We said spatial layouts of graphs were irrelevant

- We define graphs as sets of vertices and edges
- However, we'll certainly want to be able to reason about bandwidth, distance, capacity, etc. of the real world things our graph represents
  - Whether a link is 1 gigabit or 10 megabit will drastically affect our analysis of traffic flowing through a network
  - Having a road between two cities that is a 1 lane country road is very different from having a 4 lane highway
  - If two airports are 2000 miles apart, the number of flights going in and out between them will be drastically different from airports 200 miles apart

We can represent such information with edge weights

- How do we store edge weights?
  - Adjacency matrix?
  - Adjacency list?
  - Do we need a whole new graph representation?
- How do weights affect finding spanning trees/shortest paths?
  - The weighted variants of these problems are called finding the *minimum spanning tree* and the *weighted shortest path*

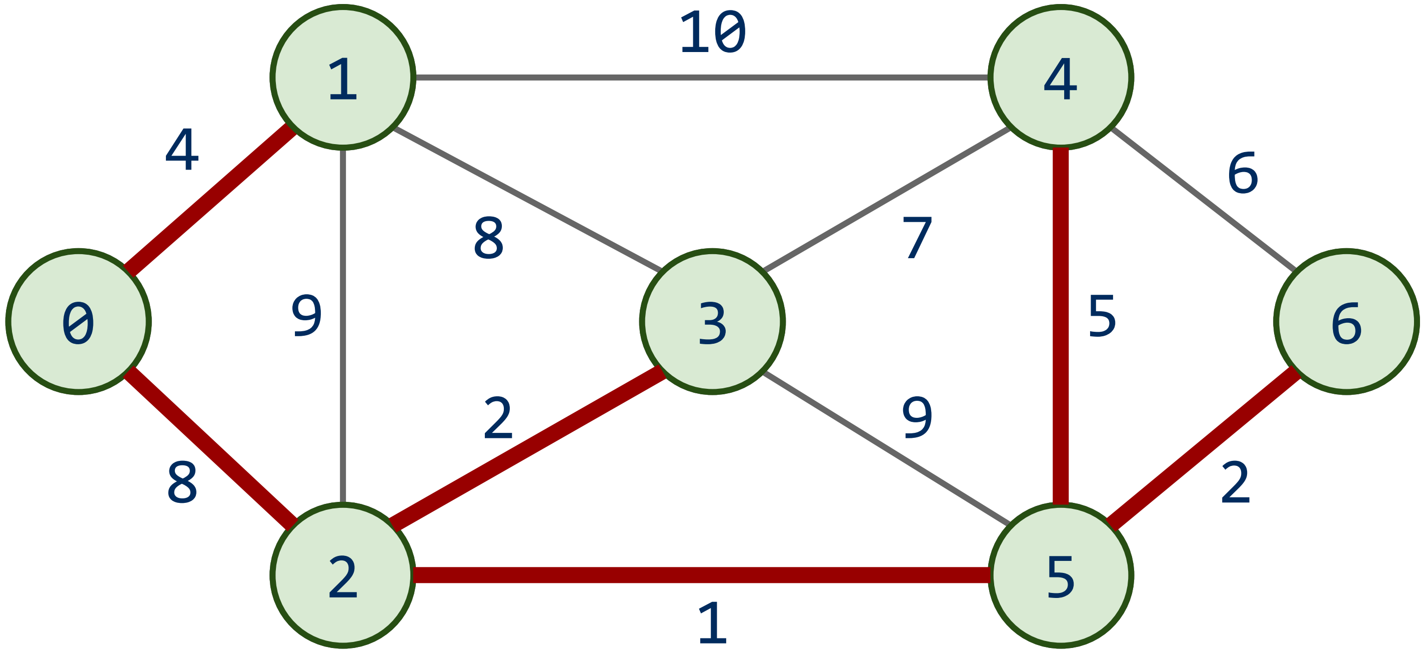
Minimum spanning trees (MST)

- Graphs can potentially have multiple spanning trees
- MST is the spanning tree that has the minimum sum of the weights of its edges

Prim's algorithm

- Initialize T to contain the starting vertex
  - T will eventually become the MST
- While there are vertices not in T:
  - Find minimum edge-weight edge that connects a vertex in T to a vertex not yet in T
  - Add the edge with its vertex to T

Prim’s algorithm



Runtime of Prim's

- At each step, check all possible edges
- For a complete graph:
  - First iteration:
    - v - 1 possible edges
  - Next iteration:
    - 2(v - 2) possibilities
      - Each vertex in T shared v-1 edges with other vertices, but the edges they shared with each other already in T
  - Next:
    - 3(v - 3) possibilities
  - …
- Runtime:
  - $\Sigma_{i\ =\ 1\ \text{to}\ v-1}\ (i * (v - i)) = \Theta(\text{largest term} * \text{number of terms})$
  - number of terms = *v-1*
  - largest term is $v^2/4$ (when $i=v/2$)
  - Evaluates to $\Theta(v^3)$